
# Tower Builder

Quick reference & Manual
v0.9.2.1 [WIP] for Editor v0.97

## 1. Introduction

The Tower Builder is a versatile tool that lets you create any tower you want for Mimic Hunter in a very intuitive way. Almost all functions we used to create the original game content are available to you to build and customize your own hunting adventure!

Though most of the actual building is done in a point-and-click way, you'll need basic scripting skills to create more advanced interactions such as dialogs, puzzles, traps, triggers, or timed events. But don't fret! Mimic Hunter uses a lightweight **LUA script** interface with only a dozen commands to remember (all documented!) beside language basics (that we won't cover here—though we'll include example scripts—, but you can find plenty online!)

You can test your tower on-the-fly with the Preview mode tool, and once finished, you can share it with other players through Steam Workshop (or by simply giving them your tower file).

Let's begin!

## 2. The Editor screen

The Editor screen has **three main parts**.

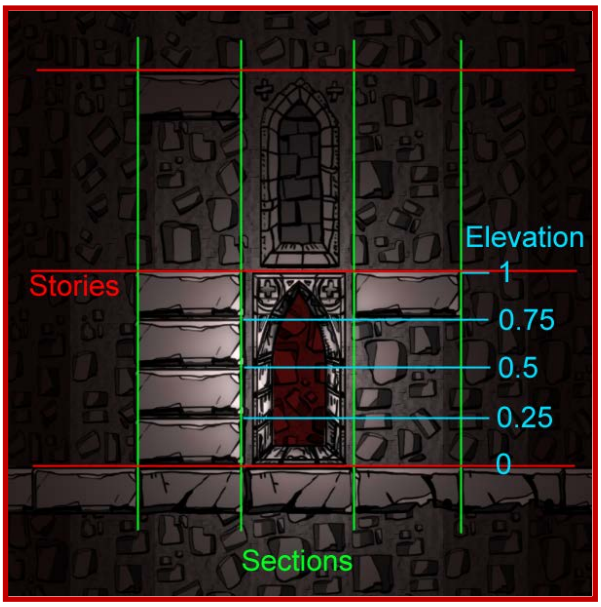
In the **middle**, you can see the tower just like you were in Play mode, and can interact with it: place, select, or remove elements, based on what tool you have selected. You can use the **horizontal movement** controls to *rotate* the tower, and the **vertical movement** controls to change your viewing *elevation*. You can also *zoom in or out* with the **mouse scroll wheel.**

On the **left** you can find the **Main toolbar**. The tools are **Walls (F1), Platforms (F2), Entities (F3), Inspector (F4)** and **Overview (F5)**. The Walls, Platforms, and Entities tools enable you to place or remove elements of the respective kind. The Inspector lets you to select any element that is already placed, and view/edit its special properties. The Overview tool is basically a list of all important elements placed, featuring quick jump-to.

On the **right** is the **Editor menu**. Most things here are quite self-explanatory: you can create a new tower, load one, save your current creation, or change global settings (such as lighting or the name of the tower). You can also enter Preview mode here (see later).

## 3. Anatomy of a tower

All towers in Mimic Hunter have two main vertical parts: the wider **"midtower"** that starts on the ground, and the leaner **"peak"** that starts on the top of the midtower. Height is primarily measured in **stories;** this is the exact height of any and all wall elements, such as doors, torches, etc. All stories have the same height, and wall elements can be only placed per exact stories, they can't be offseted (unlike platforms and entities). (The player is approximately 0.75 stories tall.) When you **create a new tower**, you have to specify the height in stories for both the midtower and the peak. The midtower can be 10-200 stories high, while the peak's height can be set between 0-200 stories, enabling you to disable it completely if you want to. **Caution!** Once a new tower has been created, you cannot modify its height!




Height offset inside a story is called **elevation**, with 0 being the bottom of the story, and 1 being the top. All platforms (at least their colliders) are **0.25 units** high. The smallest usable subdivision for positioning via the editor is **1:16** of a story's height (0.0675 units); arbitrary values can only be specified via scripting, though it is not recommended in most cases. Adjacent platforms placed with this smallest difference in elevation act as normal stairs, without the need of manual jumping. In scripting, elevation always refers to the top of a platform, but the bottom of an entity (critter or item, except bats ☺).

On the horizontal axis, stories are divided into **sections**, a.k.a. *sides*. The midtower has 24, while the peak has only 16 sections. Sections act as the "horizontal grid" for all elements, including platforms and entities—nothing can be offseted horizontally (i.e. entities are always placed at the horizontal center of a section, and all platforms and walls are exactly one section wide.)

## 4. Walls

Beside decorative elements, walls also include *doors*, *mirrors*, *torches, switches, sewers, climbable elements*, and *alcoves*.

By "wall" we always refer to **special walls**, as the tower itself is made of automatic "normal" walls with no special visual features or functions. Thus, when you "create a wall," the normal wall only becomes a special wall, and when you remove one, it reverts to a normal wall. "Void" walls are also just special walls, not the logical absence of a wall!

To **create** a wall, select the *Walls tool* (F1), then pick a wall type from the palette, and simply left-click on a section on the tower itself. You can paint over an existing wall, but any special properties you previously set (save the name) will be lost. To **remove** a wall, simply right-click on it. Hold down the **left control** key when creating a wall to auto-paint the whole story (excluding already custom-named walls) with your selection, or to remove a full story of special walls.

After you placed one or more walls you want to work with, switch to the Inspector tool (F4) and select the element you want to edit. All walls have a **name** property, but except for *doors*, *mirrors*, *torches*, *switches,* and *sewers,* its only function is to custom-identify the wall in the Overview for your convenience. Name can't be *null* or *empty*.

### 4.1. Doors and mirrors

For a door or mirror to function properly, you have to set up a few special properties. Doors and mirrors have a **target** property that identifies another door or mirror they transfer to. The target property refers to the **name** property of the target. So, for example, if you want to set up a simple two-way door, name door #1 "door_1" and door #2 "door_2", and set door #1's target property to "door_2", and door #2's to "door_1". Simple! **Caution!** Names are unique. Use a naming convention you find comfortable, yet precise enough not to get confused when you have many elements. (The Overview tool will only show walls that have a user-specified name (not the default "Wall #XX_YY"), but the default names also work as targets.)

You can also set the initial **state** of the door or mirror, which is quite self-explanatory.

During the game session, **scripts** are used to control doors, or events triggered by doors. Doors have two specific **events:** when you (try to) enter one, and when you exit on the other side. To script an enter event, create a script file named *[name of the door].lua* in the tower's script path—if it exists, it will be automatically executed every time the player tries to enter the door (if you want to limit this behavior, you have to set up your own conditions in the script). When you exit on the other side, the script *[name of the door]_d.lua* will be executed (that is, name of the door you *entered!*) To control (i.e. open/close) doors/mirrors, the **ToggleDoor()/ToggleMirror()** commands are used, see the scripting reference.

For a door to have **metal sounds** instead of wooden-like ones, its name must contain **"metal"** or **"tech"** either as a prefix or suffix!

**Important!** For a mirror to work properly, its name must contain the word "**mirror**"!

**4.2 Torches**

Torches are generally decorative-only, but it's good practice to use them to signal activations, timed events, etc. by turning them on/off.

The **name** is used to identify the torch/candle in a **ToggleLight()** command.

You can also set their initial **state**.

**4.3 Switches**

The **name** is used to identify the switch in a **ToggleSwitch()** command.

You can also set their initial **state**.

When the switch is **activated** (used), the **script** file with the name of the switch (if any) will be executed. You can also specify an inline quick script to be executed *before*/instead of the main script.

Switches automatically declare a global runtime-only LUA variable **AND** a tower-level persistent superglobal integer (accessible with *XXXMetaInt()*) *[name of the switch]_disabled = 0,* with which you can fully disable the switch if set to 1. Disabled switches don't even flip, so they'll never execute their script, not even if the toggling is executed via another script!

**4.4 Edicts**

Edicts are used to display a simple wall of text when the player interacts with them.

You can set the **mode** of the edict to either automatically display a standard dialog with the text specified when used, or to run a full script. In the **script(ure)** field, you can specify this text or script.

**4.5 Sewers**

Sewers are decorative-only by themselves.

The **name** is used to identify the sewer in a **ToggleSewer()** command.

You can also set their initial **state**.

**4.6 The tree subset**

Tree-themed walls are a special subset of the base palette that you can toggle by pressing the T key in Wall mode. While the tree subset is enabled, normal walls you place (by "deleting" a wall, i.e. right-clicking) become tree barks, and wall-void transitions you paint become wall-tree bark transitions. Other tree-themed special walls can be placed with or without the subset mode enabled; simply select them from the palette.

## 5. Platforms

To **create** a platform, select the *Platforms tool* (F2), pick the desired platform from the palette, and left-click on a wall segment at the desired elevation. To **remove** a platform, right-click on it with the Platforms tool selected.

By default, placing a platform will not paint the very exact type that is selected, but a random variant of that "family" (i.e. a group of similar looking platforms in the same row or 2 rows). This ensures that you can create good looking levels with the least effort. However, if you want to force the **exact variant** you selected, hold down the **left control** while left-clicking.

All types of platforms have the same **properties**, with only a few exceptions that behave slightly differently.

Platforms with **spikes, thorns,** or **icicles,** and **hot surfaces** will automatically do the respective amount of damage to the player if they touch it; this can't be altered. Fungal, snowy and icy platforms have different physic properties, this also can't be altered.

### 5.1 Properties

**Name** — Optional. Needed for accessing the platform from script. When the player steps on a platform, the script *[name of platform].lua* will be automatically run (if any). If a critter steps on it, *[name of platform]_a.lua* will be run (if any). When the platform is destroyed, *[name of platform]_d.lua* will be run (if any).

**Type start** — Lower bound of *type identifier* (the number seen below each platform in the palette).

**Type end** — Upper bound of *type identifier*. The platform will be a random type between *Type start* and *Type end* on every reload.

**Decay steps** — *Maximum integrity* ("health") of the platform. -1 for *indestructible*. If >0, the platform will lose 1 point from its current integrity whenever the player or a critter touches it.

**Decay current** — *Current integrity* of the platform. If lower than the maximum, cracks will be seen. *Note:* doesn't work well with non-block "platforms" like #72-73 (thorns) and #116-119 (floating stones).

**Decay wait** — Minimum *delay* in *seconds* between integrity losses. For example, with a *decay wait* value of 1, the player would have to stand on the platform for a second before it decays further. For some platforms with a DoT effect (#108-115), also controls the delay between damage ticks.

**Behavior** — For all platforms except #19 (the one with two spikes on the sides), it controls movement; for #19, it controls arrow shooting behavior. **SetBehavior()** can be used to set this behavior in run-time. Platforms have their independent internal timers, which enables you to pause a behavior by setting this value to 0, then resume it from the last position or timing when resetting it to >0. Valid values are:

**0:** default (static) — movement / shooting halted
**1:** slow *vertical* movement (9 secs for full range) / shoot *left*
**2:** medium *vertical* movement (6 secs) / shoot *right*
**3:** fast *vertical* movement (3 secs) / shoot *both ways*
**4-6:** slow/medium/fast *horizontal* movement (9/6/3 secs), respectively (ineligible for shooter)

**[Low limit]** — For moving platforms: *lower* vertical range (in elevation) / *left* horizontal range (in angles).

**[High limit]** — For moving platforms: *upper* vertical range (in elevation) / *right* horizontal range (in angles).

**[Interval]** — For arrow shooter (#19) only: *delay* in *seconds* between arrows. Be advised, don't use a too low (<0.5) value unless you absolutely know what you're doing (for example, a "controlled burst" that is stopped from script afterwards), as too many arrows will cripple performance!

**[Velocity]** — For arrow shooter (#19) only: *speed* of arrows.

**Quick script** — Additional script text to be executed (*after* the stand-alone script file, if any) when the player stands on the platform. You can add a script up to 700 characters this way. If you need more, use the .lua file of the platform's name, or run an additional file with the **Run()** function.

# Appendix A — LUA script reference

## 1. General functions

**Run** (*string* fileName)

> **fileName:** name of script file to run, *without* extension
>
> Runs the specified script file from **"[AppData]/Scripts/"**. If the given *fileName* starts with "Plot/", or does not contain "/" at all, the game will look for it in a subfolder defined by the Tower's **scriptPath** setting. Compiled (exported) tower files only "carry" the contents of their respective *scriptPath* folder, so make sure that all your custom scripts are stored there.

**Exec** (*string* script, *float* delay = 0)

> **script:** LUA script text to run
> **delay:** delay in seconds before execution
>
> Runs the given text as a LUA script (not a file!) after the specified *delay*. You can embed Run() as script to run script files with a delay, e.g. Exec('Run("*fileName*")').

*int* **GetMetaInt /** *int* **GetSaveInt** (*string* propertyName, *int* DV = -1)

> **propertyName:** name of custom tower-level / superglobal integer property (variable) to retrieve
>
> Gets the value of *propertyName*, or **DV** if it's not yet declared. For a list of built-in properties, see *Appendix B.3*.

*string* **GetMetaStr** (*string* propertyName, *string* DV = "")

> **propertyName:** name of custom global string property (variable) to retrieve
>
> Gets the value of *propertyName*, or **DV** (i.e. default value) if it's not yet declared.

**SetMetaInt / SetSaveInt** (*string* propertyName, *int* propertyValue)

> **propertyName:** name of custom tower-level / superglobal integer property (variable) to set
>
> Declares *propertyName* if it hasn't been yet, and sets its value to *propertyValue* (overwrites it if it exists.) *Note:* for all "Single tower" purposes, the two functions (and their Get methods) are practically identical!

**SetMetaStr** (*string* propertyName, *string* propertyValue)

> **propertyName:** name of custom global string property (variable) to set
>
> Declares *propertyName* if it hasn't been yet, and sets its value to *propertyValue*. (Overwrites if it exists.)

**ShowDialog** (*string* image, *string* caption, *string* sound = *nil*, *string* script = *nil*, *string* infoText = *nil*, *float* delay = 0.25, *int* d = 0)

> **image:** name of image to show in dialog, see *Appendix B.1*
> **caption:** text to display in the dialog
> **sound:** name of sound to play when the dialog appears, see *Appendix B.2*
> **script:** script text to execute when the dialog is *closed* (use *Run("xxx")* to execute a file)
> **infoText:** text to show in the bottom-screen info bar
> **delay:** delay in seconds before the dialog appears
> **d:** position of dialog; 0 = bottom half of screen, 1 = upper half of screen
>
> Shows a dialog with the specified contents & features. The game is always paused while a dialog is active. The user will either press Attack, Jump, Confirm or Cancel to close the dialog; global variable **"_action"** will store this result until the next dialog is closed: **Attack = 2**, **Jump/Confirm = 1**, **Cancel = 0**; e.g. if you define a script "myVariable = _action" to run when the dialog closes, *myVariable* will store how the dialog was closed—you can create choices this way, etc. (a good practice is to use the info bar to display what to press).

## 2. World interaction functions

**ToggleLight** (*string* name, *bool* force = *false*, *bool* value = *false*, *float* delay = 0)

> **name:** name of wall to toggle light-type component of
> **force:** *false* = state of light will be toggled (on->off, off->on); *true* = state of light will be set to *value*
> **value:** state to set light to if *force* is *true*
>
> Toggles/sets the state of the specified light-type wall.

**ToggleDoor** (*string* name, *bool* force = *false*, *bool* value = *false*, *float* delay = 0)

> **name:** name of wall to toggle door-type component of
> **force:** *false* = state of door will be toggled (open->closed, closed->open); *true* = state of door will be set to *value*
> **value:** state to set door to if *force* is *true*
>
> Toggles/sets the state of the specified door-type wall.

**ToggleMirror** (*string* name, *int* state, *float* delay = 0)

> **name:** name of wall to toggle mirror-type component of
> **state:** state to set mirror to; 0 = sleeping (inactive), 1 = awaking (inactive), 2 = awake (active)
>
> Sets the state of the specified mirror-type wall.

**ToggleSwitch** (*string* name, *bool* force = *false*, *bool* value = *false, bool* doScript = *true*, *float* delay = 0)

> **name:** name of wall to toggle switch-type component of
> **force:** *false* = state will be toggled (up->down, down->up); *true* = state will be set to *value*
> **value:** state to set switch to if *force* is *true*
> **doScript:** *true* = execute scripts on toggle like manual activation; *false* = do not execute scripts
>
> Toggles/sets the state of the specified switch-type wall.

**ToggleSewer** (*string* name, *bool* force = *false*, *bool* value = *false*, *float* delay = 0)

> **name:** name of wall to toggle sewer-type component of
> **force:** *false* = state of sewer will be toggled (on->off, off->on); *true* = state of sewer will be set to *value*
> **value:** state to set sewer to if *force* is *true*
>
> Toggles/sets the state of water from the sewer.

*int* **GetDoorState** (*string* name)

> **name:** name of door
>
> Returns the state of the specified door (0 = open; 1 = closed), or -1 if it doesn't exist.

*int* **GetMirrorState** (*string* name)

> **name:** name of mirror
>
> Returns the state of the specified mirror, as states explained at **ToggleMirror()**, or -1 if it doesn't exist.

**SetBehavior** (*string* name, *int* behavior = 0, *float* delay = 0)

> **name:** name of eligible platform
>
> Sets a platform's behavior. Use this to dynamically control "elevators", spear launchers, etc.

**SetAtmo** (*int* tier, *int* channel, *float* value)

> **tier:** 0 = ground; 1 = midtower; 2 = top
> **channel:** 0 = red; 1 = green; 2 = blue; 3 = sky exposure; 4 = HDR (i.e. highlight bloom) level
> **value:** intensity of channel, 0-1
>
> Use it to procedurally control ambient lighting. Sets the color or exposure value of the specified *channel* of the specified *tier* (values between exact tiers are extrapolated linearly), as also explained in *Tower Settings.*
>
> *Note:* changing HDR will automatically re-tune the intensity of player-centered lights to avoid over-exposure. If you experience a small change in player "brightness" under some circumstances, that is normal.

*float* **GetAtmo** (*int* tier, *int* channel)

> Retrieves an atmospheric value, as per definitions seen at *SetAtmo()*.

**SetSnowfall / SetRainfall / SetFirefall** (*bool* isOn = *true*)

> Enables (**isOn** = *true*) or disables (*false*) the respective atmospheric effect. If enables, it will also automatically disable the other two effects if active. Atmospheric effects don't have tunable parameters as of now.

## 3. Item-related functions

**SpawnItem** (*string* name, *string* image, *string* script, *int* story, *float* elevation, *int* section, *float* size = 1, *bool* interactable = *true, Vector3* worldPos = *zero* {0, 0, 0})

> **name:** name of item (for reference)
> **image:** appearance if item, i.e. name of image to display item as, see *Appendix B.1*
> **script:** script to execute when the item is picked up or touched (if it's a checkpoint)
> **story, elevation, section:** position of item, see **Chapter 3** for reference (*Note:* stories: 0-x; sections:1-24/16)
> **size:** size of item, relative to its normal size (clamped between 0.1 and 2)
> **interactable:** if *true*, item interacts with player; if *false*, it's only a decoration (no scripts will run)
> **worldPos:** if not *zero*, item will be spawned at exact coordinates in 3D space instead of standard positioning
>
> Spawns the specified item. If **name** is *nil* or *empty* (""), the item can be still picked up, but it won't appear in the inventory. If **name** starts with "checkpoint", the item will be treated as a checkpoint irrespective of **image** == "altar" or not. Also, if the item has a non-null name and you set **image** to "altar", it will become a checkpoint automatically. If **name** contains "DoT" (case sensitive!) and **interactable** *== true*, then instead of picking it up, the **script** will be run every second while the player is standing at/"in" the item. When run from the player's or a critter's *onDeath* event, you can pass **_lastPlayerPos** / **_lastCritterPos** as *worldPos* to spawn the item at the location of the (last related) death. *Items created with worldPos will instantly "fall" to the first platform below!*

**DestroyItem** (*string* name, *bool* showDebris = *true*)

> Instantly destroys the item of the given **name** in the tower (not in the inventory; see **RemoveItem()**) — use **Exec()** to make it delayed. The argument **showDebris** controls if visual & audial debris effects will be played.

*bool* **HasItem** (*string* name)

> Returns if the player has the item of the specified **name**. Always returns *true* if God mode is on in the Editor.

**RemoveItem** (*string* name) — Removes the item with the specified **name** from the inventory.

**SetItemDeadDrop** (*string* name, *string* script = *{safe neutral default; don't use empty or nil!}*)

> Sets the specified item to be dropped (i.e. *removed from inventory*) when the player dies. The specified script text will also be run (if any). Also use **SpawnItem()** with *_lastPlayerPos* as **worldPos** to actually drop an item!

**SetItemsPerma** () — Sets all items in the inventory to permanent (non-droppable).

**ToggleGargoyle** (*string* name, *bool* isWaterRunning, *float* delay = 0)

> Toggles water from a gargoyle on/off.

## 4. Miscellaneous functions

**MainMenu_Quit** ()

Returns to the Main Menu. Use it to conclude a level.

**PlaySound** (*string* id, *bool* forceRestart = *true*)

Plays a sound from the **assorted sounds** library (if not playing OR **forceRestart** == *true*), see *Appendix B.2.1.*

**SetBossMode** (*int* mode [0: none, 1: small, 2: big, 3: victory])

Sets "boss" (i.e. special game event) mode, with the appropriate musical and visual transitions.

**ShowMessage** (*string* message, *string* dialogSoundID, *string* assortedSoundID, *float* duration, *float* r, *float* g, *float* b)

Displays a message (like "Secret found!" or a countdown) at the top-center of the screen, with optional sounds, duration, and color (**r, g, b** are 0-1).

**GroundContent** (*string* id, *bool* state = *true*)

Toggles additional ground content on/off. Recommended use in startup script. For IDs, see *Appendix B.5*.

**SetBlur** (*float* size, *int* iterations = 2)

Set camera blur (interface not affected). Set **size** to 0 to disable completely.

**SetGrayscale** (*float* factor) / **SetSepiaTone** (*float* factor)

Set camera grayscale / sepia tone post effect **factor** (0-1). 0 disables the effect completely. UI not affected.

**Quake** (*int* steps = 20, *float* stepDuration = 0.1, *float* strength = 0.25)

Shakes the camera. Use **PlaySound()** to also play an appropriate sound (e.g. *"debris_1", "debris_2"*)!

## 5. Player-related functions

**AddMaxHealth** (*int* amount)

Adds **amount** extra, filled health slots ("hearts") permanently.

**AddMaxMana** (*int* amount)

Adds **amount × 25** maximum (and current) mana.

**AddDiamonds** (*int* amount)

Adds **amount** Blood diamonds. You can use a negative value to take away some.

**DoDamage** (*int* amount, *bool* blockable = *false*, *bool* bypassesEnergyShield = *true*)

Deal **amount** damage to the player instantly. You can also specify if it is blockable, and/or bypasses the energy shield or not.

**DrinkHealth** (*int* amount, *int* soundID = 2)

Instantly restores *health* of the specified **amount**, up to the maximum. Set **soundID** to -1 for no sound. (See *Appendix B.2* for internal ID of eligible sounds.)

**DrinkMana** (*int* amount, *int* soundID = 2)

Instantly restores *mana* (energy) of the specified **amount**, up to the maximum. Also see *DrinkHealth()*.

**PickupDagger** (*int* amount)

Instantly gives the specified **amount** of *daggers* to the player. Will not exceed the maximum (10).

**ReturnToGround** (*bool* lockControls = *false*)

Returns the player to the ground, outside the tower, optionally locking controls. Useful for end-level cutscenes.

**SetPlayerLight** (*bool* state)

Toggles the light centered on the player. Off is useful for creating pitch dark or heavily ambient scenes.

**Teleport** (*int* story, *float* elevation, *int* section, *bool* ignoreVelocity = *false*)

Instantly repositions the player to the specified coordinates. If **ignoreVelocity** is *true*, the player will lose all speed in the process, allowing him to avoid damage when teleporting out from a free fall, or accidentally falling off the target platform if he was moving when the teleportation commenced.

## 6. Critter-related functions

*int* **SpawnCritter** (*int* critterType:0-9, *int* story, *float* elev, *int* section)

Spawns a full health critter at the specified coordinates with default properties and returns its **internal id** for use with **SetCritter(), ReanimateId()** or **DamageCritterId()**.

**critterType:** 0 = sack; 1 = common chest; 2 = ancient chest; 3 = doombat; 4 = runic sack; 5 = fungal chest; 6 = crystallic ancient chest; 7 = skeleton; 8 = stone spawn (small); 9 = stone guardian (the big one ☺); 10 = horned heavy sack; 11 = Halloween creeper

**SetCritter** (*int* id, *float* erH = 1, *float* erV = 3, *float* fleeFactor = 0.25, int HP = -1, *bool* returnToStart = *false*, *string* onDeath = *nil*)

Sets additional properties for a critter spawned *via script*. Pass the *internal id* of the critter as **id**.

**Reanimate** (*string* name, *int* health) / **ReanimateId** (*int* id, *int* health)

Instantly resurrects the **skeleton** (critterType==7) of the given **name / id** with the given **health**. By very definition, the latter can be only used with critters created via script.

**DamageCritter** (*string* name, *int* amount) / **DamageCritterId** (*int* id, *int* amount)

Deal **amount** damage to a critter by **name / id** (returned by *SpawnCritter()*) instantly.

**SpawnStone** (*int* story, *float* elev, *int* section, *float* size = 1, *int* damage = 1)

Spawns a falling stone that can damage both the player and critters on collision (and can be avoided via hiding in an alcove). For best visual results, spawn at least two full stories higher than the player's actual position, so the stone will "fall into" the screen bounds.

## 7. Additional structure manipulation (for advanced usage)

**CreatePlatform** (*int* story, *float* elev, *int* section, *int* tier_lo = 0, *int* tier_hi = 15, *string* name = *nil*, *int* decay_steps = -1, *int* decay_current = -1, *float* decay_interval = 1, *int* behavior = 0, *float* limit_lo = 0, *float* limit_hi = 0, *string* script = *nil*)

Instantly creates a platform with the given properties.

**CreateWall** (*int* story, *int* section, *string* w, *string* name = *nil*, *int* state = 0, *string* target = *nil*)

Instantly creates a special wall with the specified properties. For *edicts*, **state** refers to *mode* (0 = text only, 1 = script); for others (doors, etc.), please see the respective **Toggle** function for state reference. The parameter **w** identifies the wall's type, as per *Appendix B.4*.

**DeletePlatform** (*string* name) — for immediate removal

Instantly deletes the platform (without any effects) of the given **name**.

**RemovePlatform** (*string* name, *bool* removeFromPlot = *true, float* delay = 0) — with visual destruction

Instantly destroys the platform of the given **name** with both visual & audio effects.

**SetDoorTarget** (*string* name, *string* target)

(Re)sets the target of a door. A door can only have one target at a given time, but with some scripting, you can easily create doors that lead to different destinations based on the circumstances. Please note, though, that the script that fires when the player passes through the door can't be used to set the target of that very transition, as it will only take effect afterwards!

## 8. LUA extensions

The following functions are based on the exposed core functions described above, and are programmed in LUA. See the "LibExtensions.lua" file for more insight & templates for your own scripts!

**FadeToDark** (*float* duration = 5, *float* stepping = 0.1, *float* factor = 0)

Fades all tiers and channels (excl. HDR) to their *current value* × **factor** over **duration** seconds, with ticks every **stepping** seconds.

**FadeBack** (*float* duration = 5, *float* stepping = 0.1)

Fades all tiers and channels (excl. HDR) back to their initial values (i.e. before the last **FadeToXXX()** call) over **duration** seconds, with ticks every **stepping** seconds. Always call a **FadeToXXX()** at least once before this!

**Countdown** (*float* from = 10, *float* to = 0, *float* stepping = -1)

Initiates a textual countdown in the center-top of the screen (via **ShowMessage()**) with tick sounds, ranging from **from** to **to,** with **stepping** *delay* and value *decrease* between each step.

**Grayscale** (*float* from = 0, *float* to = 1, *float* duration = 1)

Gradually changes *grayscale* effect value (0-1) between **from** and **to** in **duration** seconds (in 0.1 sec steps).

**SepiaTone** (*float* from = 0, *float* to = 1, *float* duration = 1)

Gradually changes *sepiatone* effect value (0-1) between **from** and **to** in **duration** seconds (in 0.1 sec steps).

**SpawnPlatform** (*float* delay, *int* story, *float* elev, *int* section, *int* tier_lo, *int* tier_hi, *string* name,
*int* decay_steps, *int* decay_current, *float* decay_interval)

Spawns a destructible platform with "popping" visual & audio effects after **delay** seconds (the actual platform emerges **0.05** seconds later). This function is best used after destroying a platform either via script or by decay (in this case call it from *[name of platform]_d.lua*) to re-spawn it with the same name, so the player can try the session again, indefinitely.

**Lightning** ()

Full-screen lightning effect with sound. Lasts 0.25 seconds. Can't be called again while in progress.

## Appendix B.1 — Assorted image (dialog/item) name reference

Use these with **ShowDialog()** or **SpawnItem()** as the ***image*** (*string*) parameter.

Items marked with * should be used in *dialogs only* (not as an actual item to be placed).
Items marked with ** are non-sprite-based special effects and are absolutely not suitable for use in a dialog.
Diamond ***: if used with *SpawnItem()*: can't be named and can't use script (set both to *nil*!), and won't be stored in saved game, but can be picked up like the normal ones dropped by critters.
Info in () is not part of the identifier. ☺

amulet_air
amulet_earth
amulet_fire
amulet_ice
amulet_void
altar
*Barnabas **
*Barnabas_ghost **
*Barnabas_undead **
Barnabas_apparition *(animated, full body)*
brazier_air
brazier_earth
brazier_fire
brazier_ice
brazier_void
book_1
book_2
boot
candelabre
crystal
dagger
*diamond ****
fungus_1_green
fungus_1_yellow
fungus_2_green
fungus_2_yellow
fungus_3_green
fungus_3_yellow
fuse_key
*fx_fire ***
*fx_frozen_mist ***
*fx_poison_cloud ***
gargoyle_1
gargoyle_2
gargoyle_3
*heart **
key_1
key_2
key_3
keystone_earth
keystone_fire
keystone_space *(void)*
keystone_water *(ice)*
keystone_wind *(air)*
mana
necklace
petri_1
petri_1_eyes
petri_1_frozen
petri_2
petri_2_frozen
petri_3
petri_4
petri_5
petri_6
petri_7
petri_8
plant_1_green
plant_1_yellow
potion
*rapier **
*warlock **
*Ratimousse **
*Ratimousse_angry **
*Ratimousse_beaten **
*Ratimousse_dead **
*Ratimousse_grinning **
*Ratimousse_scared **
*Ratimousse_smiling **
*Ratimousse_thinking **
ring
ring_air
ring_earth
ring_fire
ring_ice
ring_void
rune
scroll
statue
tissue
*X5_x_Edict_01 **
*X5_x_Edict_02 **
*X5_x_Edict_03 **

## Appendix B.2 — Sound name reference

**1. Generic (assorted) sounds**

Use these with **PlaySound()** as the ***id*** parameter, or with **ShowMessage()** as the ***assortedSoundID*** parameter.

*Internal IDs* are in (), you can use these with **DrinkXXX()** functions, for example.

altar (12)
ambient_1 (15)
ambient_2 (16)
click (7)
debris_1 (10)
debris_2 (11)
door_bulge (0)
door_bulge_metal (17)
door_close (4)
door_close_metal (19)
door_open (3)
door_open_metal (18)
door_stone (20)
door_unlock (9)
drink (2)
mirror_1 (13)
mirror_2 (14)
pickup (8)
resurrect (6)
switch (5)
switch_metal (21)
tick (22)
thunder_1 (23)
thunder_2 (24)
wind_1 (25)
wind_2 (26)

**2. Dialog sounds**

Use these with **ShowDialog()** as the ***sound*** parameter, or with **ShowMessage()** as the ***dialogSoundID*** parameter.

dark_1
dark_2
die
find
happy
idea
mystery
ominous_1
ominous_2
ominous_3
success
suspicion
victory

## Appendix B.3 — Predefined/reserved meta properties

These string/integer meta properties are not "reserved" *per se*, but are used by the game internally. Use caution when overriding them via scripting—only properties marked by * should be ever accessed for write by script, and only in special cases!

**1. Tower meta strings**

Access with **GetMetaStr()/SetMetaStr()**.

| | |
|---|---|
| GUID | *Global Unique Identifier of tower (if "forking" a level, remember to get a new GUID in Settings!)* |
| towerName | *name of tower* |
| towerAuthor | *author of tower* |
| towerDescription | *description of tower* |
| deadline * | *in seconds; stored as string for specific reasons* |
| scriptPath | *path relative to "[Mimic Hunter user data]/Scripts/"* |
| startupScript | *script file to run (from tower's script folder) when tower is first loaded (game start)* |
| onDeath * | *script text to execute when player dies. Will not override default dialog, use only for other things!* |
| onDeadline * | *script text to execute when deadline expires (Time attack & Survival only). If none, default will run.* |

**2. Tower meta integers**

Access with **GetMetaInt()/SetMetaInt()**.

| | |
|---|---|
| levelType | *0 = Short story, 1 = Time Attack, 2 = Survival* |
| difficulty | *0 = easy, 1 = normal, 2 = hard* |
| groundset | *0-4* |
| icon | *0-6* |
| snowfall | *0-1* |
| rainfall | *0-1* |
| firefall | *0-1* |
| pe_blur * | *strength of blur post-effect (0-1000, corresponds to float value of 0-10); default: 0* |
| pe_grayscale * | *strength of grayscale post-effect (0-100, corresponds to float value of 0-1 as percent); default: 0* |
| pe_sepia * | *strength of sepia post-effect (0-100, corresponds to float value of 0-1 as percent); default: 0* |
| playerLight * | 0 = off, 1 = on (default) |

*+ for each custom-named switch, treat* **[name of switch]_disabled** *as reserved.*

**3. Save integers**

Access with **GetSaveInt()/SetSaveInt()**.

| | |
|---|---|
| towerTimeElapsed * | *Note: in rare cases, you might need to modify it to add a bonus/penalty to mission time.* |

## Appendix B.4 — Wall type ID reference

Use these with **CreateWall()** as the ***w*** (*string*) parameter. Names are case-sensitive, as always.

Walls are in the same exact order as in the wall painter (left to right, top to bottom).

Sets: **X1** = frozen; **X3**, **X4** = yellow/green fungal; **X5** = sewers, edicts; **X6** = oriental; **X7** = demonic; **X9** = ruined walls

Alcove_01
Alcove_02
Alcove_03
Door_01
Door_02
Mirror
Ivy_01
Ivy_02
Ivy_03
Switch
Tentacles_01
Tentacles_02
Tentacles_03
Thorns_01
Thorns_02
Thorns_03
Torch_01
Torch_02
Torch_03
Window_01
Window_02
Window_03
Window_04
Window_05
X1_Alcove_01
X1_Alcove_02
X1_Alcove_03
X1_Door_01
X1_Door_02
Mirror_02 *(icy)*
X1_Door_04
X2_Chains
X3_Alcove_01
X3_Alcove_02
X3_Alcove_03
X3_Door_01
X3_Door_02
Mirror_03 *(fungal)*
X3_Fungi_01
X3_Fungi_02
X3_Fungi_03
X3_Fungus_large
X3_Ivy_01
X3_Ivy_02
X3_Ivy_03
X4_Fungi_01
X4_Fungi_02
X4_Ivy_01
X4_Ivy_02
X4_Ivy_03
X4_Moss_01
X4_Moss_02
X5_Sewer_01
X5_Sewer_02
X5_Sewer_03
X5_Sewer_04
X5_x_Edict_01
X5_x_Edict_02
X5_x_Edict_03
X6_Alcove_01
X6_Alcove_02
X6_Door_01
X6_Door_02
X6_Door_03
X6_Lamp_01
X6_Switch_01
X6_Switch_02
X6_Switch_03
X6_Torch_01
X6_Trellis_01
X6_Trellis_02
X6_Trellis_03
X6_Window_01
X6_Window_02
X7_Alcove_01
X7_Alcove_02
X7_Door_01
X7_Door_02
X7_Lamp_01
X7_Pipes_01
X7_Pipes_02
X7_Pipes_03
X7_Switch_01
X7_Switch_02
X7_Torch_01
X7_Torch_02
X7_Window_01
X8_Bare
X9_Door_00 *(2x2 demon gate)*
X9_Door_01 *(2x2 demon gate)*
X9_Door_10 *(2x2 demon gate)*
X9_Door_11 *(2x2 demon gate)*
X9_Wall_01
X9_Wall_02
X9_Wall_03_flip
X9_Wall_04_flip
X9_Wall_11
X9_Wall_12
X9_Wall_13_flip
X9_Wall_14_flip
X9_Wall_15
X9_Wall_16
X9_Wall_17_flip
X9_Wall_18_flip
X9_Wall_21
X9_Wall_22
X9_Wall_23_flip
X9_Wall_24_flip
X9_Wall_25
X9_Wall_26
X9_Wall_27_flip
X9_Wall_28_flip
X9_Wall_31
X9_Wall_32
X9_Wall_33
X9_Wall_34
X9_Wall_35
X9_Wall_36
X9_Wall_37
X9_Wall_38
X9_Wall_41
X9_Wall_42
X9_Wall_43
X9_Wall_44
X9_Wall_45
X9_Wall_46
X9_Wall_47
X9_Wall_48
X9_Wall_51
X9_Wall_52
X9_Wall_53
X9_Wall_54
X9_Wall_61
X9_Wall_62
X9_Wall_63
X9_Wall_64
X9_Wall_71
X9_Wall_72
X9_Wall_73
X9_Wall_74
X9_Wall_75
X9_Wall_76
X9_Wall_77
X9_Wall_78 *(void)*

## Appendix B.5 — Ground content ID reference

Use these with **GroundContent()** as the **id** parameter.

| | |
|---|---|
| **blood_tree** | A large, blossoming blood tree to the left, and a small one to the right of the tower, plus additional rocks/rubble around. (+rep #AdvJam2017)<br>Not really suitable for icy setting, but compatible with all other ground content sets. |
| **woods_west** | A large group of red-leaved trees to the West with falling leaves.<br>Don't use with frozen elements. Compatible with bare sets. |
| **woods_east** | A smaller group of red-leaved trees to the East.<br>Don't use with frozen elements. Compatible with bare sets. |
| **bare_woods_west** | A large group of bare trees to the West.<br>Compatible with any other woods or bushes. |
| **bare_woods_east** | A smaller group of bare trees to the East.<br>Compatible with any other woods or bushes. |
| **bushes** | A large number of red-leaved bushes scattered all around.<br>Don't use with frozen elements. Compatible with bare sets. |
| **bare_bushes** | A large number of bare bushes scattered all around.<br>Compatible with any other woods or bushes. |
| **frozen_woods** | A large number of frozen trees scattered all around.<br>Don't use with standard (red-leaved) elements. Compatible with bare sets. |
| **frozen_bushes** | A large number of frozen bushes scattered all around.<br>Don't use with standard (red-leaved) elements. Compatible with bare sets. |